5 Names, Bindings, and Scopes

1. 5.1 Introduction
2. 5.2 Names
3. 5.3 Variables
4. 5.4 The Concept of Binding
5. 5.5 Scope
6. 5.6 Scope and Lifetime
7. 5.7 Referencing Environments
8. 5.8 Named Constants

This chapter introduces the fundamental semantic issues of variables. It begins by describing the nature of names and special words in programming languages. The attributes of variables, including type, address, and value, are then discussed, including the issue of aliases. The important concepts of binding and binding times are introduced next, including the different possible binding times for variable attributes and how they define four different categories of variables. Following that, two very different scoping rules for names, static and dynamic, are described, along with the concept of a referencing environment of a statement. Finally, named constants and variable initialization are discussed.

5.1 Introduction

Imperative programming languages are, to varying degrees, abstractions of the underlying von Neumann computer architecture. The architecture's two primary components are its memory, which stores both instructions and data, and its processor, which provides operations for modifying the contents of the memory. The abstractions in a language for the memory cells of the machine are variables. In some cases, the characteristics of the abstractions are very close to the characteristics of the cells; an example of this is an integer variable, which is usually represented directly in one or more bytes of memory. In other cases, the abstractions are far removed from the organization of the hardware memory, as with a three-dimensional array, which requires a software mapping function to support the abstraction.

A variable can be characterized by a collection of properties, or attributes, the most important of which is type, a fundamental concept in programming languages. Designing the data types of a language requires that a variety of issues be considered. (Data types are discussed in Chapter 6.) Among the most important of these issues are the scope and lifetime of variables.

Functional programming languages allow expressions to be named. These named expressions appear like assignments to variable names in imperative languages, but are fundamentally different in that they cannot be changed. So, they are like the named constants of the imperative languages. Pure functional languages do not have variables that are like those of the imperative languages. However, many functional languages do include such variables.

In the remainder of this book, families of languages will often be referred to as if they were single languages. For example, Fortran will mean all of the versions of Fortran. This is also the case for Ada. Likewise, a reference to C will mean the original version of C, as well as C89 and C99. When a specific version of a language is named, it is because it is different from the other family members within the topic being discussed. If we add a plus sign (+) to the name of a version of a language, we mean all versions of the language

beginning with the one named. For example, Fortran 95+ means all versions of Fortran beginning with Fortran 95. The phrase **C-based languages** will be used to refer to C, Objective-C, C++, Java, and C#.1

1. We were tempted to include the scripting languages JavaScript and PHP as C-based languages, but decided they were just a bit too different from their ancestors.

5.2 Names

Before beginning our discussion of variables, the design of one of the fundamental attributes of variables, names, must be covered. Names are also associated with subprograms, formal parameters, and other program constructs. The term *identifier* is often used interchangeably with *name*.

5.2.1 Design Issues

The following are the primary design issues for names:

- Are names case sensitive?
- Are the special words of the language reserved words or keywords?

These issues are discussed in the following two subsections, which also include examples of several design choices.

5.2.2 Name Forms

A **name** is a string of characters used to identify some entity in a program.

C99 has no length limitation on its internal names, but only the first 63 are significant. External names in C99 (those defined outside functions, which must be handled by the linker) are restricted to 31 characters. Names in Java and C# have no length limit, and all characters in them are significant. C++ does not specify a length limit on names, although implementors sometimes do.

Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters (_). Although the use of underscore characters to form names was widely used in

the 1970s and 1980s, that practice is now far less popular. In the C-based languages, it has to a large extent been replaced by the so-called *camel notation*, in which all of the words of a multiple-word name except the first are capitalized, as in `myStack`.[2] Note that the use of underscores and mixed case in names is a programming style issue, not a language design issue.

2. It is called “camel” because words written in it often have embedded uppercase letters, which look like a camel’s humps.

history note

The earliest programming languages used single-character names. This notation was natural because early programming was primarily mathematical, and mathematicians have long used single-character names for unknowns in their formal notations.

Fortran I broke with the tradition of the single-character name, allowing up to six characters in its names.

All variable names in PHP must begin with a dollar sign. In Perl, the special character at the beginning of a variable’s name, `$`, `@`, or `%`, specifies its type (although in a different sense than in other languages). In Ruby, special characters at the beginning of a variable’s name, `@` or `@@`, indicate that the variable is an instance or a class variable, respectively.

In many languages, notably the C-based languages, uppercase and lowercase letters in names are distinct; that is, names in these languages are **case sensitive**. For example, the following three names are distinct in C++: `rose`, `ROSE`, and `Rose`. To some people, this is a serious detriment to readability, because names that look very similar in fact denote different entities. In that sense, case sensitivity violates the design principle that language constructs that look similar should have similar meanings. But in languages whose variable names are case-sensitive, although `Rose` and `rose` look similar, there is no connection between them.

Obviously, not everyone agrees that case sensitivity is bad for names. In C,

the problems of case sensitivity are avoided by the convention that variable names do not include uppercase letters. In Java and C#, however, the problem cannot be escaped because many of the predefined names include both uppercase and lowercase letters. For example, the Java method for converting a string to an integer value is `parseInt`, and spellings such as `ParseInt` and `parseint` are not recognized. This is a problem of writability rather than readability, because the need to remember specific case usage makes it more difficult to write correct programs. It is a kind of intolerance on the part of the language designer, which is enforced by the compiler.

5.2.3 Special Words

Special words in programming languages are used to make programs more readable by naming actions to be performed. They also are used to separate the syntactic parts of statements and programs. In most languages, special words are classified as reserved words, which means they cannot be redefined by programmers. But in some, such as Fortran, they are only keywords, which means they can be redefined.

A **reserved word** is a special word of a programming language that cannot be used as a name. There is one potential problem with reserved words: If the language includes a large number of reserved words, the user may have difficulty making up names that are not reserved. The best example of this is COBOL, which has 300 reserved words. Unfortunately, some of the most commonly chosen names by programmers are in the list of reserved words—for example, `LENGTH`, `BOTTOM`, `DESTINATION`, and `COUNT`.

In program code examples in this book, reserved words are presented in boldface.

In most languages, names that are defined in other program units, such as Java packages and C and C++ libraries, can be made visible to a program. These names are predefined, but visible only if explicitly imported. Once imported, they cannot be redefined.

5.3 Variables

A program variable is an abstraction of a computer memory cell or collection of cells. Programmers often think of variables as names for memory locations, but there is much more to a variable than just a name.

The move from machine languages to assembly languages was largely one of replacing absolute numeric memory addresses for data with names, making programs far more readable and therefore easier to write and maintain. Assembly languages also provided an escape from the problem of manual absolute addressing, because the translator that converted the names to actual addresses also chose those addresses.

A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, and scope). Although this may seem too complicated for such an apparently simple concept, it provides the clearest way to explain the various aspects of variables.

Our discussion of variable attributes will lead to examinations of the important related concepts of aliases, binding, binding times, declarations, scoping rules, and referencing environments.

The name, address, type, and value attributes of variables are discussed in the following subsections. The lifetime and scope attributes are discussed in Sections 5.4.3 and 5.5, respectively.

5.3.1 Name

Variable names are the most common names in programs. They were discussed at length in Section 5.2 in the general context of entity names in programs. Most variables have names. The ones that do not are discussed in Section 5.4.3.3.

5.3.2 Address

The **address** of a variable is the machine memory address with which it is associated. This association is not as simple as it may appear. In many languages, it is possible for the same variable to be associated with different addresses at different times during the execution of the program. For example, if a subprogram has a local variable that is allocated from the run-time stack when the subprogram is called, different calls may result in that variable having different addresses. These are in a sense different instantiations of the same variable.

The process of associating variables with addresses is further discussed in Section 5.4.3. An implementation model for subprograms and their activations is discussed in Chapter 10.

The address of a variable is sometimes called its **l-value**, because the address is what is required when the name of a variable appears in the left side of an assignment.

It is possible to have multiple variables that have the same address. When more than one variable name can be used to access the same memory location, the variables are called **aliases**. Aliasing is a hindrance to readability because it allows a variable to have its value changed by an assignment to a different variable. For example, if variables named `total` and `sum` are aliases, any change to the value of `total` also changes the value of `sum` and vice versa. A reader of the program must always remember that `total` and `sum` are different names for the same memory cell. Because there can be any number of aliases in a program, this can be very difficult in practice. Aliasing also makes program verification more difficult.

Aliases can be created in programs in several different ways. One common way in C and C++ is with their union types. Unions are discussed at length in Chapter 6.

Two pointer variables are aliases when they point to the same memory location. The same is true for reference variables. This kind of aliasing is

simply a side effect of the nature of pointers and references. When a C++ pointer is set to point at a named variable, the pointer, when dereferenced, and the variable's name are aliases.

Aliasing can be created in many languages through subprogram parameters. These kinds of aliases are discussed in Chapter 9.

The time when a variable becomes associated with an address is very important to an understanding of programming languages. This subject is discussed in Section 5.4.3.

5.3.3 Type

The **type** of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type. For example, the `int` type in Java specifies a value range of −2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

5.3.4 Value

The **value** of a variable is the contents of the memory cell or cells associated with the variable. It is convenient to think of computer memory in terms of *abstract* cells, rather than physical cells. The physical cells, or individually addressable units, of most contemporary computer memories are eight-bit units called bytes. A byte size is too small for most program variables. An abstract memory cell has the size required by the variable with which it is associated. For example, although floating-point values may occupy four physical bytes in a particular implementation of a particular language, a floating-point value is thought of as occupying a single abstract memory cell. The value of each simple nonstructured type is considered to occupy a single abstract cell. Henceforth, the term *memory cell* will mean abstract memory cell.

A variable's value is sometimes called its **r-value** because it is what is

required when the name of the variable appears in the right side of an assignment statement. To access the *r*-value, the *l*-value must be determined first. Such determinations are not always simple. For example, scoping rules can greatly complicate matters, as is discussed in Section 5.5.

5.4 The Concept of Binding

A **binding** is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. The time at which a binding takes place is called **binding time**. Binding and binding times are prominent concepts in the semantics of programming languages. Bindings can take place at language design time, language implementation time, compile time, load time, link time, or run time. For example, the asterisk symbol (`*`) is usually bound to the multiplication operation at language design time. A data type, such as **`int`** in C, is bound to a range of possible values at language implementation time. At compile time, a variable in a Java program is bound to a particular data type. A variable may be bound to a storage cell when the program is loaded into memory. That same binding does not happen until run time in some cases, as with variables declared in Java methods. A call to a library subprogram is bound to the subprogram code at link time.

Consider the following C++ assignment statement:

```
count = count + 5;
```

Some of the bindings and their binding times for the parts of this assignment statement are as follows:

- The type of `count` is bound at compile time.
- The set of possible values of `count` is bound at compiler design time.
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal `5` is bound at compiler design time.
- The value of `count` is bound at execution time with this statement.

A complete understanding of the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language. For example, to understand what a subprogram does, one must understand how the actual parameters in a call are bound to the formal parameters in its definition. To determine the current value of a variable, it may be necessary to know when the variable was bound to storage and with which statement or statements.

5.4.1 Binding of Attributes to Variables

A binding is **static** if it first occurs before run time begins and remains unchanged throughout program execution. If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic**. The physical binding of a variable to a storage cell in a virtual memory environment is complex, because the page or segment of the address space in which the cell resides may be moved in and out of memory many times during program execution. In a sense, such variables are bound and unbound repeatedly. These bindings, however, are maintained by computer hardware, and the changes are invisible to the program and the user. Because they are not important to the discussion, we are not concerned with these hardware bindings. The essential point is to distinguish between static and dynamic bindings.

5.4.2 Type Bindings

Before a variable can be referenced in a program, it must be bound to a data type. The two important aspects of this binding are how the type is specified and when the binding takes place. Types can be specified statically through some form of explicit or implicit declaration.

5.4.2.1 Static Type Binding

An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type. An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements. In this case, the first appearance of a variable name in a program constitutes its implicit declaration. Both explicit and implicit declarations create static bindings to types.

Most widely used programming languages that use static type binding exclusively and were designed since the mid-1960s require explicit declarations of all variables (Visual Basic, ML, C#, and Swift are some exceptions).

Implicit variable type binding is done by the language processor, either a compiler or an interpreter. There are several different bases for implicit variable type bindings. The simplest of these is naming conventions. In this case, the compiler or interpreter binds a variable to a type based on the syntactic form of the variable's name.

Although they are a minor convenience to programmers, implicit declarations can be detrimental to reliability because they prevent the compilation process from detecting some typographical and programmer errors.

Some of the problems with implicit declarations can be avoided by requiring names for specific types to begin with particular special characters. For example, in Perl any name that begins with `$` is a scalar, which can store either a string or a numeric value. If a name begins with `@`, it is an array; if it begins with a `%`, it is a hash structure.3 This creates different namespaces for different type variables. In this scenario, the names `@apple` and `%apple` are unrelated, because each is from a different namespace. Furthermore, a program reader always knows the type of a variable when reading its name.

3. Both arrays and hashes are considered types—both can store any scalar in their elements.

Another kind of implicit type declarations uses context. This is sometimes called **type inference**. In the simpler case, the context is the type of the value assigned to the variable in a declaration statement. For example, in C# a **`var`** declaration of a variable must include an initial value, whose type is taken as

the type of the variable. Consider the following declarations:

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

The types of `sum`, `total`, and `name` are **`int`**, **`float`**, and **`string`**, respectively. Keep in mind that these are statically typed variables—their types are fixed for the lifetime of the unit in which they are declared.

Visual Basic, Swift, and the functional languages ML, Haskell, OCaml, and F# also use type inferencing.

5.4.2.2 Dynamic Type Binding

With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment. Such an assignment may also bind the variable to an address and a memory cell, because different type values may require different amounts of storage. Any variable can be assigned any type value. Furthermore, a variable's type can change any number of times during program execution. It is important to realize that the type of a variable whose type is dynamically bound may be temporary.

When the type of a variable is statically bound, the name of the variable can be thought of being bound to a type, in the sense that the type and name of a variable are simultaneously bound. However, when a variable's type is dynamically bound, its name can be thought of as being only temporarily bound to a type. In reality, the names of variables are never bound to types. Names can be bound to variables and variables can be bound to types.

Languages in which types are dynamically bound are dramatically different from those in which types are statically bound. The primary advantage of dynamic binding of variables to types is that it provides more programming

flexibility. For example, a program to process numeric data in a language that uses dynamic type binding can be written as a generic program, meaning that it is capable of dealing with data of any numeric type. Whatever type data is input will be acceptable, because the variables in which the data are to be stored can be bound to the correct type when the data is assigned to the variables after input. By contrast, because of static binding of types, one cannot write a C program to process data without knowing the type of that data.

Before the mid-1990s, the most commonly used programming languages used static type binding, the primary exceptions being some functional languages such as Lisp. However, since then there has been a significant shift to languages that use dynamic type binding. In Python, Ruby, JavaScript, and PHP, type binding is dynamic. For example, a JavaScript script may contain the following statement:

```
list = [10.2, 3.5];
```

Regardless of the previous type of the variable named `list`, this assignment causes it to become the name of a single-dimensioned array of length 2. If the statement

```
list = 47;
```

followed the previous example assignment, `list` would become the name of a scalar variable.

The option of dynamic type binding was included in C# 2010. A variable can be declared to use dynamic type binding by including the **`dynamic`** reserved word in its declaration, as in the following example:

```
dynamic  any;
```

This is similar, although also different from declaring `any` to have type **`object`**. It is similar in that `any` can be assigned a value of any type, just as if it were declared **`object`**. It is different in that it is not useful for several different situations of interoperation; for example, with dynamically typed languages such as IronPython and IronRuby (.NET versions of Python and Ruby, respectively). However, it is useful when data of unknown type come

into a program from an external source. Class members, properties, method parameters, method return values, and local variables can all be declared **`dynamic`**.

In pure object-oriented languages—for example, Ruby—all variables are references and do not have types; all data are objects and any variable can reference any object. Variables in such languages are, in a sense, all the same type—they are references. However, unlike the references in Java, which are restricted to referencing one specific type of value, variables in Ruby can reference any object.

There are two disadvantages to dynamic type binding. First, it causes programs to be less reliable, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings. Dynamic type binding allows any variable to be assigned a value of any type. Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type. For example, suppose that in a particular JavaScript program, `i` and `x` are currently the names of scalar numeric variables and `y` is currently the name of an array. Furthermore, suppose that the program needs the assignment statement

```
i = x;
```

but because of a keying error, it has the assignment statement

```
i = y;
```

In JavaScript (or any other language that uses dynamic type binding), no error is detected in this statement by the interpreter—the type of the variable named `i` is simply changed to an array. But later uses of `i` will expect it to be a scalar, and correct results will be impossible. In a language with static type binding, such as Java, the compiler would detect the error in the assignment `i = y`, and the program would not get to execution.

Note that this disadvantage is also present to some extent in some languages that use static type binding, such as C and C++, which in many cases automatically convert the type of the RHS of an assignment to the type of the

LHS.

Perhaps the greatest disadvantage of dynamic type binding is cost. The cost of implementing dynamic attribute binding is considerable, particularly in execution time. Type checking must be done at run time. Furthermore, every variable must have a run-time descriptor associated with it to maintain the current type. The storage used for the value of a variable must be of varying size, because different type values require different amounts of storage.

Finally, languages that have dynamic type binding for variables are usually implemented using pure interpreters rather than compilers. Computers do not have instructions whose operand types are not known at compile time. Therefore, a compiler cannot build machine instructions for the expression a + b if the types of a and b are not known at compile time. Pure interpretation typically takes at least 10 times as long as it does to execute equivalent machine code. Of course, if a language is implemented with a pure interpreter, the time to do dynamic type binding is hidden by the overall time of interpretation, so it seems less costly in that environment. On the other hand, languages with static type bindings are seldom implemented by pure interpretation, because programs in these languages can be easily translated to very efficient machine code versions.

5.4.3 Storage Bindings and Lifetime

The fundamental character of an imperative programming language is in large part determined by the design of the storage bindings for its variables. It is therefore important to have a clear understanding of these bindings.

The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called **allocation**. **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

The **lifetime** of a variable is the time during which the variable is bound to a

specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. To investigate storage bindings of variables, it is convenient to separate scalar (unstructured) variables into four categories, according to their lifetimes. These categories are named static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic. In the following sections, we discuss the definitions of these four categories, along with their purposes, advantages, and disadvantages.

5.4.3.1 Static Variables

A **static variable** is one that is bound to a memory cell before program execution begins and remains bound to that same memory cell until program execution terminates. Statically bound variables have several valuable applications in programming. Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution. Sometimes it is convenient to have subprograms that are history sensitive. Such a subprogram must have local static variables.

One advantage of static variables is efficiency. All addressing of static variables can be direct;4 other kinds of variables often require indirect addressing, which is slower. Also, no run-time overhead is incurred for allocation and deallocation of static variables, although this time is often negligible.

4. In some implementations, static variables are addressed through a base register, making accesses to them as costly as for stack-allocated variables.

One disadvantage of static binding to storage is reduced flexibility; in particular, a language that has only static variables cannot support recursive subprograms. Another disadvantage is that storage cannot be shared among variables. For example, suppose a program has two subprograms, both of which require large arrays. Furthermore, suppose that the two subprograms are never active at the same time. If the arrays are static, they cannot share the same storage.

C and C++ allow programmers to include the **`static`** specifier on a variable definition in a function, making the variables it defines static. Note that when the **`static`** modifier appears in the declaration of a variable in a class definition in C++, Java, and C#, it also implies that the variable is a class variable, rather than an instance variable. Class variables are created statically some time before the class is first instantiated.

5.4.3.2 Stack-Dynamic Variables

Stack-dynamic variables are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time. For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.

As their name indicates, stack-dynamic variables are allocated from the run-time stack.

Some languages—for example, C++ and Java—allow variable declarations to occur anywhere a statement can appear. In some implementations of these languages, all of the stack-dynamic variables declared in a function or method (not including those declared in nested blocks) may be bound to storage at the beginning of execution of the function or method, even though the declarations of some of these variables do not appear at the beginning. In such cases, the variable becomes visible at the declaration, but the storage binding (and initialization, if it is specified in the declaration) occurs when the function or method begins execution. The fact that storage binding of a variable takes place before it becomes visible does not affect the semantics of the language.

The advantages of stack-dynamic variables are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic

local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. Even in the absence of recursion, having stack-dynamic local storage for subprograms is not without merit, because all subprograms share the same memory space for their locals.

The disadvantages, relative to static variables, of stack-dynamic variables are the run-time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required, and the fact that subprograms cannot be history sensitive. The time required to allocate and deallocate stack-dynamic variables is not significant, because all of the stack-dynamic variables that are declared at the beginning of a subprogram are allocated and deallocated together, rather than by separate operations.

In Java, C++, and C#, variables defined in methods are by default stack dynamic.

All attributes other than storage are statically bound to stack-dynamic scalar variables. That is not the case for some structured types, as is discussed in Chapter 6. Implementation of allocation/deallocation processes for stack--dynamic variables is discussed in Chapter 10.

5.4.3.3 Explicit Heap-Dynamic Variables

Explicit heap-dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables. The heap is a collection of storage cells whose organization is highly disorganized due to the unpredictability of its use. The pointer or reference variable that is used to access an explicit heap-dynamic variable is created as any other scalar variable. An explicit heap-dynamic variable is created by either an operator (for example, in C++) or a call to a system subprogram provided for that purpose (for example, in C).

In C++, the allocation operator, named **new**, uses a type name as its operand. When executed, an explicit heap-dynamic variable of the operand type is created and its address is returned. Because an explicit heap-dynamic variable is bound to a type at compile time, that binding is static. However, such variables are bound to storage at the time they are created, which is during run time.

In addition to a subprogram or operator for creating explicit heap-dynamic variables, some languages include a subprogram or operator for explicitly destroying them.

As an example of explicit heap-dynamic variables, consider the following C++ code segment:

```
int  *intnode;      // Create a pointer
intnode =  new int; // Create the heap-dynamic variable
. . .
delete  intnode;   // Deallocate the heap-dynamic variable
                        // to which intnode points
```

In this example, an explicit heap-dynamic variable of **int** type is created by the **new** operator. This variable can then be referenced through the pointer, `intnode`. Later, the variable is deallocated by the **delete** operator. C++ requires the explicit deallocation operator **delete**, because it does not use implicit storage reclamation, such as garbage collection.

In Java, all data except the primitive scalars are objects. Java objects are explicitly heap dynamic and are accessed through reference variables. Java has no way of explicitly destroying a heap-dynamic variable; rather, implicit garbage collection is used. Garbage collection is discussed in Chapter 6.

C# has both explicit heap-dynamic and stack-dynamic objects, all of which are implicitly deallocated. In addition, C# supports C++-style pointers. Such pointers are used to reference heap, stack, and even static variables and objects. These pointers have the same dangers as those of C++, and the objects they reference on the heap are not implicitly deallocated. Pointers are included in C# to allow C# components to interoperate with C and C++ components. To discourage their use, and also to make clear to any program reader that the code uses pointers, the header of any method that defines a

pointer must include the reserved word **`unsafe`**.

Explicit heap-dynamic variables are often used to construct dynamic structures, such as linked lists and trees, that need to grow and/or shrink during execution. Such structures can be built conveniently using pointers or references and explicit heap-dynamic variables.

The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation. This is essentially the problem of heap management, which is costly and complicated. Implementation methods for explicit heap-dynamic variables are discussed at length in Chapter 6.

5.4.3.4 Implicit Heap-Dynamic Variables

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned. For example, consider the following JavaScript assignment statement:

```
highs = [74, 84, 86, 90, 71];
```

Regardless of whether the variable named `highs` was previously used in the program or what it was used for, it is now an array of five numeric values.

The advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written. One disadvantage of implicit heap-dynamic variables is the run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler, as discussed in Section 5.4.2.2.

5.5 Scope

One of the important factors in understanding variables is scope. The **scope** of a variable is the range of statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced or assigned in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable, or in the case of a functional language, how a name is associated with an expression. In particular, scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes (blocks are discussed in Section 5.5.2). A clear understanding of these rules for a language is therefore essential to the ability to write or read programs in that language.

A variable is **local** in a program unit or block if it is declared there. The nonlocal variables of a program unit or block are those that are visible within the program unit or block but are not declared there. Global variables are a special category of nonlocal variables, which are discussed in Section 5.5.4.

Scoping issues of classes, packages, and namespaces are discussed in Chapter 11.

5.5.1 Static Scope

ALGOL 60 introduced the method of binding names to nonlocal variables called **static scoping**,[5] which has been copied by many subsequent imperative languages and many nonimperative languages as well. Static scoping is so named because the scope of a variable can be statically determined—that is, prior to execution. This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.

5. Static scoping is sometimes called *lexical scoping*.

There are two categories of static-scoped languages: those in which subprograms can be nested, which creates nested static scopes, and those in which subprograms cannot be nested. In the latter category, static scopes are also created by subprograms but nested scopes are created only by nested class definitions and blocks.

Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python allow nested subprograms, but the C-based languages do not.

Our discussion of static scoping in this section focuses on those languages that allow nested subprograms. Initially, we assume that *all* scopes are associated with program units and that all referenced nonlocal variables are declared in other program units.6 In this chapter, it is assumed that scoping is the only method of accessing nonlocal variables in the languages under discussion. This is not true for all languages. It is not even true for all languages that use static scoping, but the assumption simplifies the discussion here.

6. Nonlocal variables not defined in other program units are discussed in Section 5.5.4.

When the reader of a program finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared (either explicitly or implicitly). In static-scoped languages with nested subprograms, this process can be thought of in the following way. Suppose a reference is made to a variable `x` in subprogram `sub1`. The correct declaration is found by first searching the declarations of subprogram `sub1`. If no declaration is found for the variable there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is called its **static parent**. If a declaration of `x` is not found there, the search continues to the next-larger enclosing unit (the unit that declared `sub1`'s parent), and so forth, until a declaration for `x` is found or the largest unit's declarations have been searched without success. In that case, an undeclared variable error is reported. The static parent of subprogram `sub1`, and its static parent, and so forth up to and including the largest enclosing subprogram, are called the **static ancestors** of `sub1`. Actual implementation techniques for

static scoping, which are discussed in Chapter 10, are usually much more efficient than the process just described.

Consider the following JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function  big() {
  function  sub1() {
    var  x = 7;
    sub2();
  }
  function  sub2() {
    var  y = x;
  }
  var  x = 3;
  sub1();
}
```

Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`. This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there. The search continues in the static parent of `sub2`, `big`, where the declaration of `x` is found. The `x` declared in `sub1` is ignored, because it is not in the static ancestry of `sub2`.

In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments. For example, consider again the JavaScript function `big`. The variable `x` is declared in both `big` and in `sub1`, which is nested inside `big`. Within `sub1`, every simple reference to `x` is to the local `x`. Therefore, the outer `x` is hidden from `sub1`.

5.5.2 Blocks

Many languages allow new static scopes to be defined in the midst of executable code. This powerful concept, introduced in ALGOL 60, allows a section of code to have its own local variables whose scope is minimized. Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited. Such a

section of code is called a **block**. Blocks provide the origin of the phrase **block-structured language**.

The C-based languages allow any compound statement (a statement sequence surrounded by matched braces) to have declarations and thereby define a new scope. Such compound statements are called blocks. For example, if `list` were an integer array, one could write the following:

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

The scopes created by blocks, which could be nested in larger blocks, are treated exactly like those created by subprograms. References to variables in a block that are not declared there are connected to declarations by searching enclosing scopes (blocks or subprograms) in order of increasing size.

Consider the following skeletal C function:

```
void sub() {
  int count;
  . . .
  while (. . .) {
    int count;
    count++;
    . . .
  }
  . . .
}
```

The reference to `count` in the **`while`** loop is to that loop's local `count`. In this case, the `count` of `sub` is hidden from the code inside the **`while`** loop. In general, a declaration for a variable effectively hides any declaration of a variable with the same name in a larger enclosing scope.[7] Note that this code is legal in C and C++ but illegal in Java and C#. The designers of Java and C# believed that the reuse of names in nested blocks was too error prone to be allowed.

7. As discussed in Section 5.5.4, in C++, such hidden global variables can be accessed in the inner scope using the scope operator (::).

Although JavaScript uses static scoping for its nested functions, nonfunction blocks cannot be defined in the language.

Most functional programming languages include a construct that is related to the blocks of the imperative languages, usually named *let*. These constructs have two parts, the first of which is to bind names to values, usually specified as expressions. The second part is an expression that uses the names defined in the first part. Programs in functional languages are comprised of expressions, rather than statements. Therefore, the final part of a `let` construct is an expression, rather than a statement. In Scheme, a `let` construct is a call to the function `LET` with the following form:

- `(LET (`
- `( name1 expression1)`
- `. . .`
- `( namen expressionn))`
- expression
- `)`

The semantics of the call to `LET` is as follows: The first *n* expressions are evaluated and the values are assigned to the associated names. Then, the final expression is evaluated and the return value of `LET` is that value. This differs from a block in an imperative language in that the names are of values; they are not variables in the imperative sense. Once set, they cannot be changed. However, they are like local variables in a block in an imperative language in that their scope is local to the call to `LET`. Consider the following call to `LET`:

```
(LET (
  (top (+ a b))
  (bottom (- c d)))
  (/ top bottom)
```

`)`

This call computes and returns the value of the expression `(a + b) / (c - d)`.

In ML, the form of a **`let`** construct is as follows:

- **`let`**
- **`val`** name1 = expression1
- `...`
- **`val`** namen = expressionn
- **`in`**
- expression
- **`end`**;

Each **`val`** statement binds a name to an expression. As with Scheme, the names in the first part are like the named constants of imperative languages; once set, they cannot be changed.[8] Consider the following **`let`** construct:

8. In Chapter 15, we will see that they can be reset, but that the process actually creates a new name.

```
let
  val  top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

The general form of a **`let`** construct in F# is as follows:

- **`let`** left_side = expression

The left_side of **`let`** can be a name or a tuple pattern (a sequence of names separated by commas).